Spring 2015

Graphs of Classroom Networks

Rebecca Holliday

GRAPHS OF CLASSROOM NETWORKS

by

REBECCA HOLLIDAY

(Under the Direction of Colton Magnant)

ABSTRACT

In this work, we use the Havel-Hakimi algorithm to visualize data collected from students to investigate classroom networks. The Havel-Hakimi algorithm uses a recursive method to create a simple graph from a graphical degree sequence. In this case, the degree sequence is a representation of the students in a classroom, and we use the number of peers with whom a student studied or collaborated to determine the degree of each. We expand upon the Havel-Hakimi algorithm by coding a program in Matlab that generates random graphs with the same degree sequence. Then, we run another algorithm to find the isomorphism classes within the randomly generated graphs. Once we have reduced the problem to the isomorphism classes, we then choose a graph we think most accurately describes the classroom network. At the end of this work, we will make a note on the rainbow connection number in oriented graphs with diameter 2.

Key Words: Havel-Hakimi, graphs, degree sequence, isomorphism classes

2009 *Mathematics Subject Classification*: 90C35, 05C60, 05C80, 05C85, 05C90

GRAPHS OF CLASSROOM NETWORKS

by

REBECCA HOLLIDAY

B.S. in Applied Mathematics

A Thesis Submitted to the Graduate Faculty of Georgia Southern University in Partial Fulfillment

of the Requirement for the Degree

MASTER OF SCIENCE

STATESBORO, GEORGIA

2015

GRAPHS OF CLASSROOM NETWORKS

by

REBECCA HOLLIDAY

Major Professor: Colton Magnant

Committee: Jonathan Hilpert
Colton Magnant
Hua Wang

Electronic Version Approved:

Spring 2015

ACKNOWLEDGMENTS

I would like to thank my committee, Dr. Jonathan Hilpert, Dr. Colton Magnant, and Dr. Hua Wang, for all of their help and support. I wish to acknowledge the National Science Foundation (NSF) for funding this research (NSF TUES Type 1 REC-1245081). This funding has allowed us the opportunity to look at a common problem in a new way.

Contents

CHAPTER 1

CLASSROOM DATA GRAPHS

1.1 Introduction

The goal of this work is to create an accurate graphical representation of student interactions in a classroom. For this pursuit, we develop a method to analyze survey data gathered from two research universities. The students of these classes report the number of interactions with fellow classmates, and we use this information to create a degree sequence where each student is a vertex and the number of classmates a student typically collaborates with during classroom activities is the degree. Using this information, we take that degree sequence and use it to create classroom graphs. Note, for the purposes of this work, the graphs are not directed, meaning the relationship between students is mutual (Student A collaborates with Student B if and only if Student B collaborates with Student A).

The data are gathered from engineering students attending two universities, who participate in a twenty minute survey. This survey begins by asking how many students with whom the participant collaborated in the classroom and outside the classroom. After reporting the degree sequence, students rate the instructors teaching style (i.e. lecture, group work, etc) using a Likert scale rating system. Student responses to these questions allow us to determine the kind of instructions the professor gives to the class.

These answers are particularity important because we believe the teaching style greatly influences the overall structure graph of student relationships while in the classroom. For instance, in a classroom setting where students listen to the professor lecture, we expect to see Figure 2.1(a). However, in a classroom where the professor assigns problems to groups, we expect to see a graph resembling Figure 2.1(b).

(a) Lecture class (b) Collaborative classroom

Figure 1.1: Expected graphs for different class styles

The collaborative classroom is particularly difficult to estimate since there are unknown factors that influence group interaction, which will be discussed later.

Next, we gather information on what the students experience in different aspects of the course. These questions help us identify contribution level of each group member. The survey then gives group scenarios for the students to compare similarities to the scenario in their group's behavior. These comparisons give us a chance to see if the group works on in-depth the problems that take a lengthy amount of time to resolve or if solutions are quick to find (which will influence the connected-ness of the group in the graph).

Next, we ask questions pertaining to the individual. Here we learn several traits that tell us how motivated the student is to learning the course material and how much the student enjoys the course and/or is challenged by the course. This information we believe will help us in ensuring we find the best fitting graph to the data.

It should be noted that, as with any survey, some reports had to be removed. These are typically students who took the survey in an improbable amount of time or tried to take the survey several times.

In order to develop graphs from our classroom degree sequences, we first need to ensure the degree sequences taken from the survey data are graphical. To check if the sequence is graphical, we use criteria proven by Erdös and Gallai [10] in their

theorem:

Theorem 1. *A sequence of non-negative integers $d_1 \geq \ldots \geq d_n$ can be represented as the degree sequence of a finite simple graph on n vertices if and only if $d_1 + \ldots + d_n$ is even and*

$$\sum_{i=1}^{k} d_i \leq k(k-1) + \sum_{i=k+1}^{n} min(d_i, k)$$

holds for $1 \leq k \leq n$.

If necessary, the degree sequence is altered to make it graphical. The methods for altering the degree sequence will be explained later in Section 2.3. Once the sequence is graphical, the next goal is to find all possible graphs that could represent the degree sequence. Rather, we attempt to answer the equivalent question, how many isomorphism classes can be generated from a given degree sequence?

A graph G is said to be *isomorphic* to another graph H if there exists a bijective map $\varphi : V(G) \to V(H)$ such that $(a, b) \in E(G)$ if and only if $(\varphi(a), \varphi(b)) \in E(H)$. This equivalence relationship is denoted by $G \cong H$. The equivalence classes with respect to isomorphism are a called *isomorphism classes*. Note, that a degree sequence can have more than one isomorphism class, meaning there can be more than one graph that can represent a degree sequence.

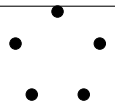
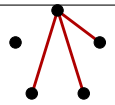
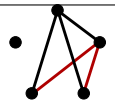
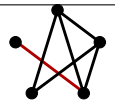
Since the number of isomorphism classes (or, if preferred, the number of non-isomorphic graphs up to isomorphism) from a given degree sequence is not easy to calculate, we implement the following two procedures, Randomizing Havel-Hakimi and The Isomorphism Algorithm, in Matlab to estimate the number of isomorphism classes. These procedures can be seen in more detail in Section 2.4.

Randomizing Havel-Hakimi stems from programs written by Gergana Bounova [2] and the Havel-Hakimi algorithm [11, 12] stating:

Theorem 2 (Havel (1955), Hakimi (1962))**.** *Let $S = (d_1, \ldots, d_n)$ be a finite list of*

nonnegative integers that is nonincreasing. List S is graphic if and only if the finite list $S' = (d_2 - 1, d_3 - 1, \ldots, d_{d_1+1} - 1, d_{d_1+2}, \ldots d_n)$ has nonnegative integers and is graphic.

To visualize the algorithm, consider the degree sequence $\{3, 2, 3, 3, 1\}$. Then, by the Havel-Hakimi Algorithm, we have:

Remaining Sequence	Sorted Sequence	Graph
$\{3, 2, 3, 3, 1\}$	$S = \{3, 3, 3, 2, 1\}$	
$\{0, 2, 2, 1, 1\}$	$S' = \{2, 2, 1, 1, 0\}$	
$\{0, 1, 0, 1, 0\}$	$S'' = \{1, 1, 0, 0, 0\}$	
$\{0, 0, 0, 0, 0\}$	$S''' = \{0, 0, 0, 0, 0\}$	

As seen above, the placement of edges is specific, however, other choices could have been made. Notice the Havel-Hakimi Algorithm will produce a single graph for a given degree sequence.

Randomizing Havel-Hakimi expands the above algorithm by allowing the program to randomly pick plausible edges. This procedure generates k random graphs of the same degree sequence, where k is a sufficiently large natural number determined by the user. These k randomly generated graphs may be isomorphic or be in separate isomorphism classes (if more than one isomorphism class exists).

From here, we use the second procedure, The Isomorphism Algorithm, created by Dharwadker and Tevet [7], to determine the number of isomorphism classes in the k random graphs. This procedure compares two graphs to determine if they are

isomorphic or not isomorphic in polynomial time. From here, we need to pick which of these isomorphism classes we think is the best fit to the classroom. At the end of Chapter 2, we will mention the method we think might be best in determining which isomorphism class is the best fit.

1.2 Is this Graphical?

Unfortunately, some of the data is not graphical by the Havel-Hakimi requirements. This is caused by either students reporting the wrong number of ingroup peers or by not having all students in the class take the survey, or some combination of the above. Hence, further modifications are needed to make the sequence graphical.

In degree sequences where not all students had participated in the survey, several tests had to be run. The degree sequences sometimes have an odd sum. In this case, we need to decide to either add a student (one we could assume existed somewhere in the classroom outside of the gathered information) or remove a student (an outlier in the degree sequence).

The first test we run is a median test. This tells us the median of the degree sequence. The purpose of this test is to add another hypothetical student with the median degree. We choose not to use the mean test since the results would frequently not be integers. Another benefit to adding a student is that it occasionally helped us avoid deleting students who took the survey. Sometimes students who had shown up as outliers in the original degree sequence would no longer be significant outliers in the new sequence with the additional hypothetical student. These situations are usually when we do not have enough students to meet the degree sequence requirements, but we are very close. However, in some situations, this technique is not enough. In the situations where adding another student does not make the degree sequence graphical, we perform a normality test to see which degrees are outliers. In these cases, the

outliers report such a high degree that we would have to add an unjustifiable amount of hypothetical students. Since the median test does not work in these scenarios, we have to remove any student who is more than 2 standard deviations away from center. In some cases, removing outliers makes the degree sequence graphical. However, in other cases, the first process, the median test, should be performed again after removing outliers to make an even sum.

1.3 Procedures

Once the data is graphical, we can start finding all the isomorphism classes using Matlab code A.1, which calls Randomizing Havel-Hakimi and The Isomorphism Algorithm. To demonstrate these algorithms, consider the graphical sequence $\{2, 2, 2, 1, 1\}$ with vertices $\{v_1, v_2, v_3, v_4, v_5\}$ respectively.

Randomizing Havel-Hakimi. For this procedure, Matlab code A.3, begin by constructing a graph G_1 using the following algorithm.

Select a vertex n_1 such that it is the first highest degree in the nonincreasing sequence. For this example, $n_1 = v_1$. Now, to add an edge to the graph, randomly select a target vertex n_2 that satisfies the following:

1. $n_2 \neq n_1$ (no self-loops).
2. An edge (n_2, n_1) does not already exist (no double/parallel edges).
3. remdeg$(n_2) > 0$, where remdeg(n_2) is the entry of the current degree sequence corresponding to n_2.

Without loss of generality, let $n_2 = v_2$. Then, we place an edge from v_1 to v_2 and update the sequence to $\{1, 1, 2, 1, 1\}$. Now, we begin the process again. since v_3 has the highest degree, $v_3 = n_1$. Again, randomly select n_2 with the same criteria above.

Continue this process until the graph G_1 has been constructed (the sum of the degree sequence is zero).

If the last edge cannot be placed, then the algorithm A.6 will be used to rewire edges. In this event, there will be a vertex x and a vertex y each with remaining degree 1 that have an existing edge between them. Let $A = V(G) \setminus \{x, y\}$ and let $B = N(x) \cup N(y)$ (shown in gray in Figure 2.2), where $N(x)$ is the set of all vertices that neighbor x. Suppose there exists an edge (s, t) such that $s, t \in A \setminus B$. Note, if (s, t) does not exist, then the algorithm stops and A.3 starts over. Then, the algorithm will randomly choose between the following two cases:

1. The edges (x, s) and (y, t) will be added (in green) and the edge (s, t) will be removed (in purple in Figure 2.2).

2. The edges (x, t) and (y, s) will be added (in red) and the edges (s, t) will be removed (again, in purple).

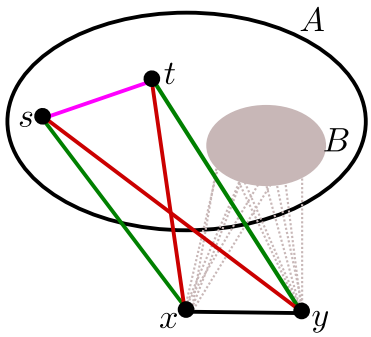


Figure 1.2: Cases for rewire algorithm

This completes the rewiring algorithm and completes G_1. This process is repeated to find G_2. Since we have two graphs to compare, we can move on to Procedure 2 to check if they are isomorphic.

The Isomorphism Algorithm. In this procedure, A.8, we determine which of the graphs found in Randomizing Havel-Hakimi are isomorphic. For simplicity,

we will run this example on two graphs that could have been found in Randomizing Havel-Hakimi, G_A and G_B, with the above degree sequence to determine if $G_A \cong G_B$.

(a) G_A (b) G_B

Figure 1.3: Two isomorphic graphs

First, using Dijkstra's Algorithm [8], we will find the length of the shortest paths between all vertices in G_A. Begin with the collateral graph $G_A \setminus v_1v_2$ (G_A with edge (v_1, v_2) removed). Note, the edge from (v_1, v_2) does not exist in G_A, but that is ok. Now, we will find the lengths of the shortest paths from $v_1 \to v_i$ for all i and $v_2 \to v_j$ for all j.

From the table to the right, we know the shortest distance is $\mathrm{d}(v_1, v_2) = 2$. We will denote the shortest distance from u to v in $G_A \setminus uv$ by sd(u,v). Note that $\mathrm{sd}(u, u) = 0$. Also, if the graph is not connected, hence there is no path from u to v, then $\mathrm{sd}(u, v) = \infty$.

Also, we can make the pair graph $G_{v_1v_2}$ by removing any vertices in the collateral graph that are not contained in the shortest path(s) from v_1 to v_2 and vice versa. For this example, $G_{v_1v_2}$ is as

Start/End	Path
$v_1 \to v_1$	(v_1)
$v_1 \to v_2$	(v_1, v_3, v_2)
$v_1 \to v_3$	(v_1, v_3)
$v_1 \to v_4$	(v_1, v_3, v_2, v_4)
$v_1 \to v_5$	(v_1, v_5)
$v_2 \to v_2$	(v_2)
$v_2 \to v_1$	(v_2, v_1)
$v_2 \to v_3$	(v_2, v_3, v_1)
$v_2 \to v_4$	(v_2, v_3, v_1)
$v_2 \to v_5$	(v_2, v_3, v_1, v_5)

follows:

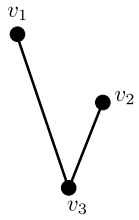


Figure 1.4: Pair Graph $G_{v_1v_2}$ for G_A

For all the pairs (u, v) in G_A, we say the (u, v)-sign, denoted s_{uv}, is as follows:

$$s_{uv} = (\pm)\mathrm{sd}(u, v).|V(G_{uv})|.|E(G_{uv})|$$

where the leading binary sign is positive if the edge $(u, v) \in E(G_A)$ and negative if $(u, v) \notin E(G_A)$. In the above example, $s_{12} = -2.3.2$. Now, we compute all s_{ij} to make the sign matrix S.

S_A	v_1	v_2	v_3	v_4	v_5
v_1	-0.1.0	-2.3.2	∞.5.3	-3.4.3	∞.5.3
v_2	-2.3.2	-0.1.0	∞.5.3	∞.5.3	-3.4.3
v_3	∞.5.3	∞.5.3	-0.1.0	-2.3.2	-2.3.2
v_4	-3.4.3	∞.5.3	-2.3.2	-0.1.0	-4.5.4
v_5	∞.5.3	-3.4.3	-2.3.2	-4.5.4	-0.1.0

We make the frequency table F_A by sorting the unique signs listed in S_A in lexicographical order and counting the occurrence of each sign in S_A.

Now, we want the rows of F_A to be in lexicographical order, denote this sorted matrix F_A^* . To do this, we need to switch columns v_1 and v_3.

The corresponding swap needs to be in S_A to make S_A^*. To do this, we swap both the columns v_1 and v_3 and the rows v_1 and v_3.

F_A	v_1	v_2	v_3	v_4	v_5
-4.5.4	0	0	0	1	1
-3.4.3	1	1	0	1	1
-2.3.2	1	1	2	1	1
-0.1.0	1	1	1	1	1
∞.5.3	2	2	2	1	1

F_A^*	v_3	v_2	v_1	v_4	v_5
-4.5.4	0	0	0	1	1
-3.4.3	0	1	1	1	1
-2.3.2	2	1	1	1	1
-0.1.0	1	1	1	1	1
∞.5.3	2	2	2	1	1

We call S_A^* the canonical representation of S_A. Repeat the previous process to find S_B^* and F_B^*.

Now the algorithm checks the statement $F_A^* = F_B^*$. If this statement is false, $F_A^* \neq F_B^*$, then $G_A \ncong G_B$. However, the converse is not true. Hence, if $F_A^* = F_B^*$, then G_A and G_B might be isomorphic or might not be. In our example, $F_A^* = F_B^*$, however, $S_A^* \neq S_B^*$. To continue investigating if $G_A \cong G_B$, the algorithm then tries to find an explicit isomorphism.

To do this, the algorithm compares each entry of S_A^* to the respective entry in S_B^*. When two entries do not match, the algorithm begins row-swapping S_B^* (and column swapping since the S matrices are symmetric) until the mismatched entry matches. Then, it repeats this process looking for other mismatched entries. After performing the swap, two scenarios can occur. Either S_B^* cannot be written as a permutation of S_A^* meaning $G_A \ncong G_B$. Or S_B^* can be written as a permutation of S_A^*

S_A^*	v_3	v_2	v_1	v_4	v_5
v_3	-0.1.0	∞.5.3	∞.5.3	-2.3.2	-2.3.2
v_2	∞.5.3	-0.1.0	-2.3.2	-3.4.3	∞.5.3
v_1	∞.5.3	-2.3.2	-0.1.0	∞.5.3	-3.4.3
v_4	-2.3.2	-3.4.3	∞.5.3	-0.1.0	-4.5.4
v_5	-2.3.2	∞.5.3	-3.4.3	-4.5.4	-0.1.0

F_B^*	v_3	v_2	v_1	v_4	v_5
-4.5.4	0	0	0	1	1
-3.4.3	0	1	1	1	1
-2.3.2	2	1	1	1	1
-0.1.0	1	1	1	1	1
∞.5.3	2	2	2	1	1

(which in our example is true), thus $G_A \cong G_B$.

In summary, we begin by producing a random graph by using procedure 1. We then produce a second graph with procedure 1, and check if it is isomorphic to the first with procedure 2. If it is, we do not store the new graph, rather we increase the multiplicity of the graph that it was isomorphic to. If it is not isomorphic to the first graph, then we store it. Next we move on to generate a third graph and compare it to the stored graphs. We continue this process for k iterations, where k is sufficiently large to increase our confidence that all isomorphism classes appear in our list of stored graphs.

Overall, for this example ($\{2, 2, 2, 1, 1\}$), it can be easily checked that there are 2 isomorphism classes and they are: a Hamiltonian Path, and a disconnected graph where the components are a K_3 and K_2. Using the previously mentioned procedures, we found 2 isomorphism classes for $k = 100$. More specifically, the multiplicity showed

S_B^*	v_3	v_2	v_1	v_4	v_5
v_3	-0.1.0	∞.5.3	∞.5.3	-2.3.2	-2.3.2
v_2	∞.5.3	-0.1.0	-2.3.2	∞.5.3	-3.4.3
v_1	∞.5.3	-2.3.2	-0.1.0	-3.4.3	∞.5.3
v_4	-2.3.2	∞.5.3	-3.4.3	-0.1.0	-4.5.4
v_5	-2.3.2	-3.4.3	∞.5.3	-4.5.4	-0.1.0

83 Hamiltonian Paths and 17 disconnected graphs.

1.4 Results of Procedures

Although the procedures we use to find the number of isomorphism classes are effective in helping us find graphs to represent the classroom, the algorithms themselves can be improved. For instance, recall in Section 2.5 that the randomly generated graphs were not equally distributed between the two isomorphism classes. The program more readily found the Hamilton Path instead of the disconnected graph.

This phenomena can be easily explained, but not easily corrected. When looking more closely at the degree sequence $\{2, 2, 2, 1, 1\}$, we see that after the first few selections for edges have been made, the graph is already destined to be a specific isomorphism class. Consider the following scenario. Recall that the program will start with v_1 when generating a random graph. Now, the program has a 50% chance of choosing to connect with a vertex of degree 2 and a 50% chance of choosing to connect with a vertex of degree 1.

Case 1. If the program picks a vertex of degree 2, the same probabilities kick in. The program, again, will either add an edge with a vertex that was originally degree 2 or degree 1.

Subcase 1. Suppose again, another vertex originally of degree 2 has been chosen.

In this case, have 33% chance of selecting a vertex originally of degree 2 and a 66% chance of selecting a vertex originally of degree 1. This next selection is important to which isomorphism class the graph will be in. Until this point, we could be in either of the two classes. Now, if the program chooses a vertex originally of degree 2, then we have a K_3, and the final graph will be in the disconnected graph isomorphism class. If the program selects a vertex originally of degree 1 then we will get a Hamiltonian Path.

Subcase 2. Now suppose the program selects one of the vertices originally with degree 1. Then, we already know the graph will be a Hamiltonian path.

Case 2. As stated, the algorithm will be with v_1, which is of degree 2. If the program then randomly picks one of the degree 1 vertices in the list, the graph will be a Hamiltonian Path. Then, after only one edge has been placed in the graph, we automatically know what type of graph will be produced.

From the above, we see that the algorithm has a much higher chance of making a Hamiltonian Path than the disconnected graph. This was confirmed by our multiplicity output which showed us that the Hamilitonian Path had been randomly generated 83 times and the disconnected graph only 17 times. Although these numbers will slightly fluctuate with each run, we expect the distribution to resemble the Hamiltonian Path being randomly generated approximately $\frac{11}{12}$ of the time, and the disconnected graph being randomly generated only approximately $\frac{1}{12}$ of the time.

It should also be noted that, unfortunately, our implementation of the previously mentioned procedures cannot run on directed graphs. We plan on improving upon this, as well as the overall efficiency of the algorithms by implementing them in a faster language.

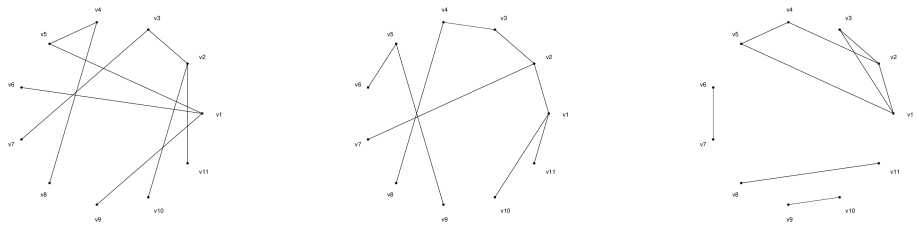
1.5 Putting it all together

The output of our program is several isomorphism classes. We now turn to the problem of deciding which of those isomorphism classes to use as the representative of the classroom.

A common scenario that happens frequently in the real world can be modeled with a K_3, which represents an easily visualized scenario. If Person A knows Person B and Person B knows Person C, then we can almost assume that in a classroom (especially of small class size) that Person A and Person C know each other. Although it might be expected that this real world scenario would be inherited in classrooms, classroom interactions may not always have many copies of K_3.

There exist scenarios where, in the above example, Person A and Person C did not know each other. Hence, creating multiple induced P_3 subgraphs in our classroom graph. One way this happens is in the following scenario: a professor tells the students to work with one other student. Another day, he gives the same instruction but the previously consulted peer is absent that day. Another scenario is if the professor specifies who the students should work with, such as "work with the student to your left." Once we knew such scenarios existed, we realized we cannot just select the graph containing the most K_3 subgraphs as an accurate representation.

Now, consider the following three graphs (each from a different isomorphism class) of an actual classroom:



Clearly, there are more isomorphism classes than the three shown above for that

degree sequence. However, it is important to note that there are clear structural differences in these particular graphs. For instance, consider the sizes of each group of students represented by the graphs. In the graph on the left, there are two groups: one containing six students and the other containing five students. Turning to the middle graph, there are also two groups, but one contains eight students and the other contains three. Lastly, the graph on the right, contains four distinct groups: three groups each with two students and another with five students. Since the isomorphism classes can have significant differences, we need an efficient method for choosing the best representation of the data.

We conjecture that anaylzing the random walks on these graphs will give us an indication of the classroom structure and help us in narrowing down our set of graphs which we consider realistic. Selecting one of these graphs to represent a real classroom, with unpredictable elements influencing its structure, is not an easy task. We also plan to add a method of estimating the missing students who did not participate in the survey.

CHAPTER 2

NOTE ON RAINBOW CONNECTION IN ORIENTED GRAPHS WITH DIAMETER 2

2.1 Introduction

The concept of rainbow connection was first introduced by Chartrand et al. in [4]. A path in an edge-colored graph is called *rainbow* if no two edges in the path receive the same color. The *rainbow connection number* of a graph is the minimum number of colors needed to color the edges of the graph so that there is a rainbow path between every pair of vertices. This and the more general rainbow k-connection number have been heavily studied in recent years in [3, 4, 5, 6, 13, 14, 16] and many other works. In particular, see [15] for a survey of results in the area.

A tournament T is an oriented complete graph. We consider only *k-strongly connected* (or simply *k-strong*) tournaments, meaning that there are k internally disjoint directed paths from each vertex to every other vertex. A directed path between two vertices in an edge-colored tournament is called *rainbow* if no two edges have the same color within the path. If there is a directed rainbow path between every pair of vertices in a graph, then the coloring is called *rainbow connected*. The smallest number of colors needed for a tournament to be rainbow connected is called the (directed) *rainbow connection number*, denoted $\overrightarrow{rc}(T)$. The diameter d of a tournament is the largest, over all ordered pairs of vertices, number of edges in the shortest path between the two vertices.

In [9], the following theorem was proven.

Theorem 3 (Dorbec et al. [9])**.** *For any tournament T of diameter d,*

$$d \leq \overrightarrow{rc}(T) \leq d + 2.$$

The authors noted that $d + 2$ may not be the best upper bound.

Question 1. *For each diameter d, is $d+1$ or $d+2$ the sharp upper bound on $\overrightarrow{rc}(T)$ where T has diameter d.*

We believe that a $(d+1)$-coloring is possible, at least in some cases. Indeed, we show that for tournaments of diameter 2, this improved upper bound holds.

Theorem 4. *For any tournament T of diameter 2,*

$$2 \leq \overrightarrow{rc}(T) \leq 3.$$

The proof of this result is provided in Section 2.2.

More generally, we initiate the study of the rainbow k-connection number of a tournament. An edge-colored tournament is called *rainbow k-connected* if, between every pair of vertices, there is a set of k internally disjoint rainbow paths. The *rainbow k-connection number* of a tournament, denoted $\overrightarrow{rc}_k(T)$, is then the minimum number of colors needed to produce a rainbow k-connected coloring of the tournament T. To state our next result, we let the *k-total-diameter*, denoted $d_k(T)$, be the maximum (over all pairs of vertices) of the smallest number of edges in a set of k internally disjoint paths between the vertices.

Theorem 5. *Given an integer $k \geq 2$ and a tournament T of order n with $d_k(T) = d$,*

$$\overrightarrow{rc}_k(T) \leq \frac{d}{1-(1-\frac{1}{n^2})^{1/d}}.$$

The proof of Theorem 5 is an easy application of the probabilistic method and is presented in Section 2.3.

Next we define some more notation. Say a set of k internally disjoint paths from a vertex x to a vertex y is *minimum* if the longest path in the set is as short as possible, over all such sets of paths. Let the *k^{th} diameter* denote the maximum length, over all pairs of vertices u, v, of the longest path in a minimum set of k internally disjoint

$u - v$ paths. More formally, if $\ell_k(u, v)$ is the minimum length of the longest path in a set of k internally disjoint $u - v$ paths, then the k^{th} diameter of a graph G is

$$\max_{u,v \in V(G)} \ell_k(u, v).$$

Note that the 1^{st} diameter is simply the diameter of the graph. Also note that the k^{th} diameter is at least $\frac{d_k(T)}{k}$. Our final result considers tournaments with small k^{th} diameter and provides a bound on the rainbow connection number.

Theorem 6. *A strongly connected tournament T of k^{th} diameter 2 has $\overrightarrow{rc}_k(T) \leq 3 + k + 2\binom{k}{2}$.*

The proof of Theorem 6 is presented in Section 2.4. This naturally leads to the following problem.

Problem 1. *Produce sharp bounds on $\overrightarrow{rc}_k(T)$ in terms of the k^{th} diameter of T.*

2.2 Proof of Theorem 4

The sharpness of the upper bound is given by the following example. Let A be a directed triangle and let u and v be single vertices. Direct all edges from v to A, from A to u and from u to v. Any 2-coloring of this graph must color two edges of A with a single color. This induces a directed monochromatic P_3. Let a_1 be the initial vertex of this P_3 and let a_3 be the terminal vertex and note that the edge between a_1 and a_3 is directed from a_3 to a_1. See Figure 2.1.

The only possible rainbow path from a_1 to a_3 must pass through u and v, meaning that it must use 3 different colors. Thus, this graph has diameter 2 but rainbow connection number 3. Larger graphs with the same property can be built by replacing vertices with directed triangles and blowing up edges in the natural way.

We now prove Theorem 4.

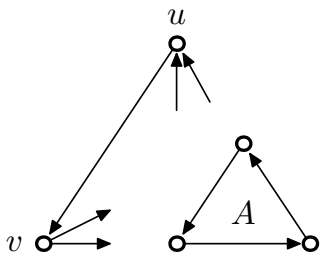


Figure 2.1: A tournament T with diameter 2 and rainbow connection number 3.

Proof. Let T be a tournament of diameter 2. Let $a \to b \to c$ be a shortest path from a vertex a to another vertex c. Let A_1 denote the out-neighborhood of a and let $A_2 = T \setminus (A_1 \cup \{a\})$. Note that A_i is the set of vertices at distance i from a and, in particular, $b \in A_1$ and $c \in A_2$. Color all edges from a to A_1 with color 1 (red in Figure 2.2). All edges from vertices in A_1 to vertices in A_2 have color 2 (blue), and all edges from vertices in A_2 to the vertex a have color 3 (green). All edges from vertices in A_2 to vertices in A_1 also have color 3. Finally, all edges within the same set, either A_1 or A_2, have color 1. This coloring is similar to the one used by Dorbec et al. in [9] to prove Theorem 3.

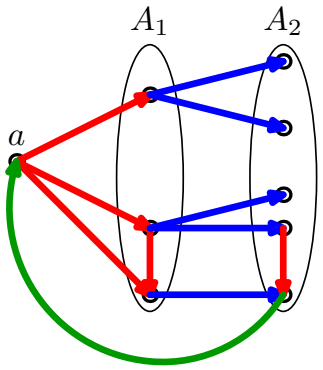


Figure 2.2: Coloring of the tournament.

In order to show that this coloring is rainbow connected, we consider cases based on the location of two selected vertices x and y and find rainbow paths between them.

If $x = a$, we trivially find a rainbow path to y for any choice of $y \in A_1$ since A_1 is the out-neighborhood of a. If $y \in A_2$, then by construction, there is a rainbow path containing some vertex $w \in A_1$ such that $x \to w \to y$ with colors 1 and 2 respectively.

If $y = a$ and $x \in A_2$, the result is again trivial since a is an out-neighbor of every vertex in A_2. Also, if $x \in A_1$, then again there is a rainbow path of length 2, namely $x \to w \to y$ for some $w \in A_2$ using colors 2 and 3.

If $x \in A_2$ and $y \in A_1$, then, by construction, there is a rainbow path $x \to a \to y$ using colors 3 and 1, respectively.

Finally, suppose $x \in A_1$ and $y \in A_2$. If the edge $x \to y$ is in $E(T)$, then there is a trivially rainbow path of length 1. Since the diameter is 2, there exists a vertex with $x \to w \to y$. Regardless of the location of w, this path is rainbow by construction. More specifically, if $w \in A_1$, then the path $x \to w \to y$ uses colors 1 and 2, respectively. If $w \in A_2$, then the path $x \to w \to y$ exists uses colors 2 and 1, respectively. Hence, every tournament with diameter 2 has rainbow connection number at most 3. □

2.3 Proof of Theorem 5

For this proof, we use the probabilistic method as described by Alon and Spencer in [1]. This bound is likely far from the best possible, particularly when n is much bigger than d, and we make little effort to optimize it.

We now prove Theorem 5.

Proof. Consider a tournament on n vertices and set $c = \frac{d}{1-(1-\frac{1}{n^2})^{1/d}}$. Label the vertices of T with $\{v_1, v_2, \ldots, v_n\}$. Randomly color the edges of T using c colors. Let $X_{i,j}$ be an indicator variable which takes the value 1 if there is no set of k internally disjoint rainbow paths from v_i to v_j. Since there is a set of such paths on at most d edges, we

compute the expectation of $X_{i,j}$ to be

$$E(X_{i,j}) = 1 - \frac{c!}{(c-d+1)!c^d} < 1 - \left(1 - \frac{d}{c}\right)^d.$$

By linearity of expectation, if we set $X = \sum_{i,j} X_{i,j}$, we get

$$E(X) = \sum E(X_{i,j}) \leq 2\binom{n}{2}\left(1 - \left(1 - \frac{d}{c}\right)^d\right).$$

Since $c = \frac{d}{1-(1-\frac{1}{n^2})^{1/d}}$, we see that $E(X) < 1$ so, by the probabilistic method, there is a coloring of T with c colors that is rainbow k-connected. □

2.4 Proof of Theorem 6

For a tournament T of k^{th} diameter 2, we use the following coloring. Select a k-subset of vertices $A := \{v_1, v_2, \ldots, v_k\}$. Now, let the set of all out-neighbors of A be called A_1 and color all edges of the form $A \to A_1$ with the color C_1 and edges of the form $A_1 \to A$ with color C_2. Let $N_i' \subseteq A_1$ be the out-neighborhood of v_i for each $1 \leq i \leq k$. Define sets

$$N_i = N_i' \setminus (\cup_{j<i} N_j)$$

for all $1 \leq i \leq k$. We use one distinct color C_{N_i} on all edges within each set N_i for all i for a total of k colors. Use at most $2\binom{k}{2}$ distinct colors to color the remaining edges of A_1 such that edges of the form $N_i \to N_j$ have a different color from those of the form $N_j \to N_i$ for all $i \neq j$. This uses a total of $2\binom{k}{2} + k$ colors to color A_1.

Let A_2 be the set of all remaining vertices so that A_2 is in the out-neighborhood of A_1 and A is in the out-neighborhood of A_2. It should be noted that $A_2 = \emptyset$ is allowed. Color all edges of the form $A_1 \to A_2$ with color C_2 and the edges from $A_2 \to A$ with color C_3. All edges from A to A_2 have color C_1 and all edges from A_2 to A_1 have color C_3. The edges within the set A are colored using $\binom{k}{2}$ of the colors

previously used between sets in A_1. The edges within the set A_2, if they exist, are allowed to be any color except C_3. Hence $3 + k + 2\binom{k}{2}$ colors are used to color T.

In order to show that this coloring is rainbow connected, we consider cases based on the location of vertices x and y and find k rainbow paths from x to y.

If $x = v_i$ and $y = v_j$ for some $i \neq j$, then there are k internally disjoint paths of length at most 2 that each must be one the following: $x \to y$, $x \to N_i \to y$, or $x \to v_h \to y$ for some $h \neq i, j$. All paths of these forms are rainbow by construction.

If $x, y \in A_2$ then there are k internally disjoint paths of length 3 that each must be one of the following: $x \to v_i \to N_i \to y$, or $x \to v_i \to N_j \to y$ for all i with $1 \leq i \leq k$ and for some $j \neq i$. All such paths are rainbow by construction.

If $x \in A_1$ and $y = v_i$ then there are k internally disjoint rainbow paths of length at most 2, each having one the following forms: $x \to v_k \to y$, or $x \to w \to y$, where $w \in A_1$ and $w = y$ is allowed. If $x = v_i$ and $y \in A_1$ then there exist k internally disjoint rainbow paths of the form $x \to w \to y$ where each $w \in A_1$ and $w = y$ is allowed.

If $x = v_i$ and $y \in A_2$ then there are k internally disjoint rainbow paths of the form $x \to A_1 \to y$. If $x \in A_2$ and $y = v_i$ then there exist k internally disjoint rainbow paths $x \to w \to y$ where each w is anywhere and $w = y$ is allowed.

If $x \in A_1$, say $x \in N_i$, and $y \in A_2$, then there are easily k internally disjoint rainbow paths from x to y as in previous cases. If $x \in A_2$ and $y \in A_1$ then there are k internally disjoint rainbow paths such that $x \to v_i \to N_i \to y$ for all i.

Finally, suppose $x, y \in A_1$. If $x \in N_i$ and $y \notin N_i$, then there are k internally disjoint rainbow paths of the form $x \to w \to y$, where w is anywhere. Now let $x, y \in N_i$. Since there are k internally disjoint paths of length at most 2 from $x \to v_i$, there exist k out-neighbors of x that are outside N_i. Call this set N_x. Since the diameter is 2, there are k internally disjoint paths of the form $N_x \to w \to y$ where w

can be anywhere. Hence, there are k internally disjoint rainbow paths from x to y of the form $x \to N_x \to w \to y$.

This completes the proof of Theorem 6. □

2.5 Concluding Remarks

Unfortunately our method used in the proofs of Theorems 4 and 6 does not extend to tournaments of diameter larger than 2 so the question of a sharp result in Question 1 and Problem 1 remains open even for diameter 3. The bottleneck is clearly going from vertices in A_1 to vertices in A_2 as defined in the proofs.

REFERENCES

[1] N. Alon and J. H. Spencer. *The probabilistic method*. Wiley-Interscience Series in Discrete Mathematics and Optimization. John Wiley & Sons, Inc., Hoboken, NJ, third edition, 2008. With an appendix on the life and work of Paul Erdös.

[2] G. Bounova. Octave Networks Toolbox First Release. Zenodo. 2014. DOI: 10.5281/zenodo.10778

[3] Y. Caro, A. Lev, Y. Roditty, Z. Tuza, and R. Yuster. *On rainbow connection.* Electron. J. Combin., 15(1):Research paper 57, 13, 2008.

[4] G. Chartrand, G. L. Johns, K. A. McKeon, and P. Zhang. *Rainbow connection in graphs*. Math. Bohem., 133(1):8598, 2008.

[5] L. Chen, X. Li, and H. Lian. *Nordhaus-Gaddum-type theorem for rainbow connection number of graphs*. Graphs Combin., 29(5):12351247, 2013.

[6] D. Dellamonica, Jr., C. Magnant, and D. M. Martin. *Rainbow paths*. Discrete Math., 310(4):774781, 2010.

[7] A. Dharwadker, J.T. Tevet. *The graph isomorphism algorithm*. The Structure Semiotics Research Group S.E.R.R. Eurouniversity Tallinn. 2009.

[8] E.W. Dijkstra, *A note on two problems in connection with graphs.* Numerische Math. 1, 269-271, 1959.

[9] T. P. Dorbec, I. Schiermeyer, E. Sidorowicz, and E. Sopena. *Rainbow connection in oriented graphs*. Manuscript.

[10] P. Erdös, T. Gallai, *Gráfok elört fokszámú pontokkal*, Matematikai Lapok 11: 264274, 1960.

[11] Hakimi, S. L. , *On realizability of a set of integers as degrees of the vertices of a linear graph. I*, Journal of the Society for Industrial and Applied Mathematics 10: 496506, 1962.

[12] Havel, V. , *A remark on the existence of finite graphs*, Časopis pro pěstování matematiky (in Czech) 80: 477480, 1955.

[13] A. Kemnitz, J. Przybylo, I. Schiermeyer, and M. Wózniak. *Rainbow connection in sparse graphs*. Discuss. Math. Graph Theory, 33(1):181192, 2013.

[14] X. Li, M. Liu, and I. Schiermeyer. *Rainbow connection number of dense graphs*. Discuss. Math. Graph Theory, 33(3):603611, 2013.

[15] X. Li, Y. Shi, and Y. Sun. *Rainbow connections of graphs: a survey*. Graphs Combin., 29(1):138, 2013.

[16] I. Schiermeyer. *Bounds for the rainbow connection number of graphs*. Discuss. Math. Graph Theory, 31(2):387395, 2011.

Appendix A

MATLAB CODE

These are the codes for the previously mentioned procedures.

A.1 classesFromDegreeSequence

```
% Finds the isomorphism classes of a degree sequence.
%
% INPUT: A degree sequence d and number of iterations k.
% OUTPUT: Adjacency Matrix for each class and the
%         multiplicity of each class.
%
% Note, the matrices in the output are in nonincreasing degree
% order. Although it is NOT required for this program,
% the degree sequence input should also be in nonincreasing
% order if the output needs to correspond to the input
% vertex numbering.
%
% Modified: March 21, 2015

function [A, mults] = classesFromDegreeSequence(d, k)

% First checks if d is graphical
d = d(find(d));
if ~isGraphic(d)
  disp('The sequence is not graphic.');
  return
```

```
end

A = randomGraphFromDegreeSequence(d);
[~,ind] = sort(-sum(A));
A = A(ind, ind);
n = length(A);

[F, Sd, Sn, Sm] = sgnFreq(A);
classes = 1;
mults = [1];

for i = 1:k

% Optional wait bar so user can watch progress
  waitbar(i/k);
  M = randomGraphFromDegreeSequence(d);
  newclass = true;

  % Check if we have a new class
  for j = 1:classes
    if isIsom(A(:,:,j), M)
      mults(j) += 1;
      newclass = false;
      break;
    end
  end
```

```
    if newclass
      [~,ind] = sort(-sum(M));
      M = M(ind, ind);
      classes += 1;
      A(:,:,classes) = M;
      mults = [mults 1];
    end

end
end
```

A.2 isGraphic

```
% Checks whether a degree sequence is graphical.
%
% Source: Erds, P. and Gallai, T. "Graphs with Prescribed
% Degrees of Vertices" [Hungarian]. Mat. Lapok. 11, 264-274,
% 1960.
%
% INPUT: a sequence (vector) of numbers
% OUTPUT: boolean, true or false
%
% Note: Not generalized to directed graph degree sequences.
% Modified: March 21, 2015

function B = isGraphic(seq)

if not(isempty(find(seq<=0))) || mod(sum(seq),2)==1
    % there are non-positive degrees or their sum is odd
    B = false; return;
end

n=length(seq);
seq=-sort(-seq);  % sort in decreasing order

for k=1:n-1
    sum_dk = sum(seq(1:k));
```

```
    sum_dk1 = sum(min([k*ones(1,n-k);seq(k+1:n)]));

    if sum_dk > k*(k-1) + sum_dk1; B = false; return; end

end

B = true;
```

A.3 randomGraphFromDegreeSequence

```
% Constructing a random graph based on a given degree sequence.
% Source: Molloy M. & Reed, B. (1995) Random Structures
% and Algorithms 6, 161-179
%
% INPUT: a graphic sequence of numbers, 1xn
% OUTPUT: adjacency matrix of resulting graph, nxn
%
% Note: The simple version of this algorithm gets stuck about
% half of the time, so in this implementation the last
% problematic edge is rewired.
%
% Other routines used: adj2edgeL.m, rewireThisEdge.m,
% and edgeL2adj.m
% Modified: March 21, 2015



function adj= randomGraphFromDegreeSequence(Nseq)


niter = 0;
stubs=Nseq;                    % assign degrees to stubs
adj = zeros(length(Nseq)); % initialize adjacency matrix
perm = [2:length(stubs) 1];
```

```
old_sum = 0;
cnt=0;

while sum(stubs)>0    % while stubs are left to connect

  % catch problem graphs (debugging purposes)
  niter += 1;
  if niter > 1000
    fprintf('failed after 1000 iterations\n');
    stubs
    adj
    return;
  end

  if sum(stubs)==2 && cnt>length(stubs)
  % rewire the last edge when stuck

    el = adj2edgeL(adj);
    ind = find(stubs>0);

    if length(ind) == 1;
      elr = rewireThisEdge([el; ind(1) ind(1) 1],
                           ind(1),ind(1));
    end
    if length(ind) == 2;
```

```
    elr = rewireThisEdge([el; ind(1) ind(2) 1;
                          ind(2) ind(1) 1],
                          ind(1),ind(2));
  end

  if isempty(elr) % restart algorithm completely
    fprintf('Starting over\n');
    stubs = Nseq;
    adj = zeros(length(Nseq));
    old_sum = 0;
    cnt=0;

  else
    adj = edgeL2adj(elr);
    % return matrix with last edge rewired
    return

  end

end


new_sum = sum(stubs);

if old_sum==new_sum
  cnt = cnt+1;
```

```
    stubs = stubs(perm);
    adj = adj(perm, perm);
  end % no new nodes have been connected, counter+1

  if old_sum~=new_sum; cnt=0; end
  % new connections, restart count



  [~,n1] = max(stubs);
  % pick the node with highest number of remaining stubs

  old_sum = sum(stubs);

  ind = find(stubs.*(1-adj(n1,:))); % no double edges
  ind = ind(ind ~= n1); % no self-loops
  if length(ind) == 0; continue; end
  n2 = ind(randi(length(ind)));

  adj(n1,n2)=1; adj(n2,n1)=1;
  stubs(n1) = stubs(n1) - 1;
  stubs(n2) = stubs(n2) - 1;

end
```

A.4 adj2edgeL

```
% Convert adjacency matrix (nxn) to edge list (mx3)
%
% INPUT: adjacency matrix: nxn
% OUTPUT: edge list: mx3
%
% GB: last updated, Sep 24, 2012

function el=adj2edgeL(adj)

n=length(adj); % number of nodes
edges=find(adj>0); % indices of all edges

el=[];
for e=1:length(edges)
   [i,j]=ind2sub([n,n],edges(e)); % node indices of edge e
   el=[el; i j adj(i,j)];
end
```

A.5 edgeL2adj

```
% Convert edge list to adjacency matrix.
%
% INPUT: edge list: mx3, m - number of edges
% OUTPUT: adjacency matrix nxn, n - number of nodes
%
% Note: information about nodes is lost: indices only (i1,...in)
%       remain
%
% GB: last updated, Sep 25, 2012

function adj=edgeL2adj(el)

nodes=sort(unique([el(:,1) el(:,2)])); % get all nodes, sorted
adj=zeros(numel(nodes));               % initialize adjacency matrix

% across all edges
for i=1:size(el,1);
  adj(find(nodes==el(i,1)),find(nodes==el(i,2)))=el(i,3);
end
```

A.6 rewireThisEdge

```
% Degree-preserving rewiring of 1 given edge.
% Note: Assume unweighted undirected graph.
%
% INPUT: edge list, el (mx3) and the two nodes of the
%        edge to be rewired.
% OUTPUT: rewired edge list, same size and same degree
%         distribution
%
% Note: There are cases when rewiring is not possible
%       while simultaneously keeping the graph simple,
%       so an empty edge list is returned.
%
% Other routines used: edgeL2adj.m, kneighbors.m
% GB: last updated, Oct 25, 2012

function el = rewireThisEdge(el,i1,i2)

% check whether the edge can actually be rewired
adj = edgeL2adj(el);
neighbors = [i1, i2];
neighbors = [neighbors kneighbors(adj,i1,1)];
neighbors = [neighbors kneighbors(adj,i2,1)];

disjoint_edges = [];
```

```
for e=1:length(el)
  if (sum(ismember(neighbors,el(e,1)))==0 &&
      sum(ismember(neighbors,el(e,2)))==0)
    disjoint_edges = [disjoint_edges; el(e,:)];
  end
end

if isempty(disjoint_edges)
  errmsg = strcat('rewireThisEdge(): cannot rewire this',
                  ' graph without adding a double edge',
                  ' or a loop\n');
  fprintf(errmsg);
  el = [];
  return
end

[~,row] = ismember([i1 i2 1],el,'rows');
ind = [row];
edge1=el(ind(1),:);


% pick a random second edge from the disjoint edges
randind = randi([1,size(disjoint_edges,1)]);
edge2=disjoint_edges(randind,:);

[~,ind2] = ismember([edge2(1) edge2(2) 1],el,'rows');
```

```
ind = [ind ind2];


if rand<0.5

  % first possibility: (e11,e22) & (e12,e21)
  el(ind(1),:)=[edge1(1),edge2(2),1];
  el(ind(2),:)=[edge1(2),edge2(1),1];

  % add the symmetric equivalents
  [~,inds1] = ismember([edge1(2),edge1(1),1],el,'rows');
  el(inds1,:)=[edge2(2),edge1(1),1];

  [~,inds2] = ismember([edge2(2),edge2(1),1],el,'rows');
  el(inds2,:)=[edge2(1),edge1(2),1];

else

  % second possibility: (e11,e21) & (e12,e22)
  el(ind(1),:)=[edge1(1),edge2(1),1];
  el(ind(2),:)=[edge1(2),edge2(2),1];

  % add the symmetric equivalents
  [~,inds1] = ismember([edge1(2),edge1(1),1],el,'rows');
  el(inds1,:)=[edge2(1),edge1(1),1];
```

```
    [~,inds2] = ismember([edge2(2),edge2(1),1],el,'rows');
    el(inds2,:)=[edge2(2),edge1(2),1];

end
```

A.7 kneighbors

```
% Finds the number of k-neighbors (k links away) for every node
%
% INPUT: adjacency matrix (nxn), start node index,
%        k - number of links
% OUTPUT: vector of k-neighbors indices
%
% GB: last updated, Oct 7 2012

function kneigh = kneighbors(adj,ind,k)

adjk = adj;
for i=1:k-1; adjk = adjk*adj; end;

kneigh = find(adjk(ind,:)>0);
```

A.8 isIsom

```
% Checks if two graphs are isomorphic.
%
% Dharwadker, Tevet. (2009). "The Graph Isomorphism Algorithm"
%
% INPUT: Adjacency matrix of two graphs.
% OUTPUT: True or False
%
% Modified: March 21, 2015

function isIsom = isIsom(G1, G2)

[F1, d1, n1, m1] = sgnFreq(G1);
[F2, d2, n2, m2] = sgnFreq(G2);

isIsom = true;
[F1,sig1] = sortrows(F1);
[F2,sig2] = sortrows(F2);

if prod(size(F1) == size(F2))==0 || (F1 ~= F2)
  isIsom = false;
  return
end

d1 = d1(sig1, sig1);
```

```
n1 = n1(sig1, sig1);
m1 = m1(sig1, sig1);

d2 = d2(sig2, sig2);
n2 = n2(sig2, sig2);
m2 = m2(sig2, sig2);

% Find first mismatch
d0 = find(~(d1==d2));
n0 = find(~(n1==n2));
m0 = find(~(m1==m2));

niter = 0;

while(~(isempty(d0) && isempty(n0) && isempty(m0)))

  if niter > length(d1)^2
    disp('isIsom failed')
    return
  end
  niter += 1;

  % Find an element to swap with mismatch
  misMatch = min([d0; n0; m0]);
  [row, col] = ind2sub([length(d1), length(d1)], misMatch);
  matchd = find(d2(row, col:end) == d1(row, col));
```

```
    matchn = find(n2(row, col:end) == n1(row, col));
    matchm = find(m2(row, col:end) == m1(row, col));
    matches = intersect(intersect(matchd, matchn), matchm);

    % Nothing to swap to, graphs are not isomorphic
    if isempty(matches)
      isIsom = false;
      return
    end

    newCol = min(matches) + col - 1;

    % Swap the rows and columns of the sign matrix of graph 2
    d2(:,[col, newCol]) = d2(:,[newCol, col]);
    n2(:,[col, newCol]) = n2(:,[newCol, col]);
    m2(:,[col, newCol]) = m2(:,[newCol, col]);
    d2([col, newCol],:) = d2([newCol, col],:);
    n2([col, newCol],:) = n2([newCol, col],:);
    m2([col, newCol],:) = m2([newCol, col],:);

    % Find first mismatch
    d0 = find(~(d1==d2));
    n0 = find(~(n1==n2));
    m0 = find(~(m1==m2));

end
```

A.9 sgnFreq

```
% Creates the Sign Frequency Matrix and Sign Matrix of a graph.
%
% Dharwadker, Tevet. (2009). "The Graph Isomorphism Algorithm"
%
% INPUT: Adjacency matrix of a graph.
% OUTPUT: Sign Frequency Matrix and Sign Matrix
%
% Modified: March 21, 2015

function [F, Sd, Sn, Sm] = sgnFreq(G)

n = length(G(:,1));
Sd = zeros(n);
Sn = zeros(n);
Sm = zeros(n);
s = zeros(n*(n-1)/2, 3);
k = 1;

for i=1:n
    for j=i:n
        [Sd(i,j),Sn(i,j),Sm(i,j)] = SUV(G,i,j);
        s(k,:) = [Sd(i,j),Sn(i,j),Sm(i,j)];
        k += 1;
    end
```

```
end

Sd = Sd + Sd';
Sn = Sn + Sn' - eye(n);
Sm = Sm + Sm';

s = unique(sortrows(s),'rows')
r = length(s(:,1));
F = zeros(n,r);
for i = 1:n
    a = [Sd(:,i), Sn(:,i), Sm(:,i)];
    for j = 1:r
        for l = 1:n
            if a(l,:) == s(j,:)
                F(i,j) += 1;
            end
        end
    end
end
```

A.10 SUV

```
% Creates the entries for the Sign Matrix of a graph.
%
% Dharwadker, Tevet. (2009). "The Graph Isomorphism Algorithm"
%
% INPUT: Adjacency matrix of a graph, coordinates of sign matrix
% OUTPUT: The values of the sign matrix
%
% Modified: March 21, 2015

function [d,n,m] = SUV(G,u,v)

Guv = G;
sgn = (-1)^(Guv(u,v) + 1);
Guv(u,v) = 0;
Guv(v,u) = 0;
ind = simpleDijkstra(Guv, u);
d = ind(v);
ind = ind + simpleDijkstra(Guv, v);
ind = find(ind == simpleDijkstra(Guv, u)(v));
Guv = Guv(ind,ind);
n = length(Guv(:,1));
m = sum(sum(Guv))/2;

d = sgn*d;
```

A.11 simpleDijkstra

```
% Implementation of a simple version of the Dijkstra shortest
% path algorithm. Returns the distances from a single vertex
% to all others, doesn't save the path
%
% INPUTS: adjacency matrix, adj (nxn), start node s (index
%         between 1 and n)
% OUTPUTS: shortest path length from the start node to all
%          other nodes, 1xn
%
% Note: Works for a weighted/directed graph.
% GB: last updated, September 28, 2012

function d = simpleDijkstra(adj,s)

n=length(adj);
d = inf*ones(1,n); % distance s-all nodes
d(s) = 0;     % s-s distance
T = 1:n;     % node set with shortest paths not found yet

while not(isempty(T))
    [dmin,ind] = min(d(T));
    for j=1:length(T)
        if (adj(T(ind),T(j))>0 &&
            d(T(j))>d(T(ind))+adj(T(ind),T(j)))
```

```
            d(T(j))=d(T(ind))+adj(T(ind),T(j));
        end
    end
    T = setdiff(T,T(ind));
end
```